1
Getting Started and Working with Shell Scripting

If you work with Linux, you will come across the shell. It's usually the first program you work with. **Graphical user interface** (**GUI**) usage has become very popular due to its ease of use. Those who want to take advantage of the power of Linux will use the shell program by default:

- The shell is a program that provides the user with direct interaction with the operating system. Let's understand the stages in the evolution of the Linux operating system. Linux was developed as a free and open source substitute for the Unix OS. The chronology was as follows: The Unix operating system was developed by Ken Thomson and Dennis Ritchie in 1969. It was released in 1970. They rewrote Unix using C language in 1972.
- In 1991, Linus Torvalds developed the Linux kernel for the free operating system.

In this chapter, we will cover the following topics:

- Comparison of shells
- Working in shell
- Learning basic Linux commands
- Our first script—`Hello World`
- Compiler and interpreter–differences in processes
- When not to use scripts
- Various directories
- Working more effectively with shell–basic commands
- Working with permissions

Comparison of shells

Initially, the Unix OS used a shell program called the **Bourne shell**. Then, eventually, many more shell programs were developed for different flavors of Unix. The following is some brief information about different shells:

- `sh`—Bourne shell
- `csh`—C shell
- `ksh`—Korn shell
- `tcsh`—enhanced C shell
- `bash`—GNU Bourne Again shell
- `zsh`—extension to `bash`, `ksh`, and `tcsh`
- `pdksh`—extension to `ksh`

A brief comparison of various shells is presented in the following table:

Feature	Bourne	C	TC	Korn	Bash
Aliases	no	yes	yes	yes	yes
Command-line editing	no	no	yes	yes	yes
Advanced pattern matching	no	no	no	yes	yes
Filename completion	no	yes	yes	yes	yes
Directory stacks (`pushd` and `popd`)	no	yes	yes	no	yes
History	no	yes	yes	yes	yes
Functions	yes	no	no	Yes	yes
Key binding	no	no	yes	no	yes
Job control	no	yes	yes	yes	yes
Spelling correction	no	no	yes	no	yes
Prompt formatting	no	no	yes	no	yes

What we see here is that, generally, the syntax of all these shells is 95% similar.
In this book, we are going to follow Bash shell programming.

Tasks done by the shell

Whenever we type any text in the shell Terminal, it is the responsibility of the shell (`/bin/bash`) to execute the command properly. The activities done by the shell are as follows:

- Reading text and parsing the entered command
- Evaluating meta-characters, such as wildcards, special characters, or history characters
- Process io-redirection, pipes, and background processing
- Signal handling
- Initializing programs for execution

We will discuss the preceding topics in the subsequent chapters.

Working in the shell

Let's get started by opening the Terminal, and we will familiarize ourselves with the bash shell environment:

1. Open the Linux Terminal and type in:

```
$ echo $SHELL
/bin/bash
```

2. The preceding output in the Terminal says that the current shell is `/bin/bash`, such as the Bash shell:

```
$ bash -version
GNU bash, version 4.3.48(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
http://gnu.org/licenses/gpl.html

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Hereafter, we will use the word `Shell` to signify the Bash shell only. If we intend to use any other shell, then it will be specifically mentioned by name, such as `KORN` and other similar shells.

In Linux, filenames in lowercase and uppercase are different; for example, the files `Hello` and `hello` are two distinct files. This is unlike Windows, where case does not matter.

As far as possible, avoid using spaces in filenames or directory names such as:

- Wrong filename—`Hello World.txt`
- Correct filename—`Hello_World.txt` or `HelloWorld.txt`

This will make certain utilities or commands fail or not work as expected, for example, the `make` utility.

While typing in filenames or directory names of the existing files or folders, use the tab completion feature of Linux. This will make working with Linux faster.

Learning basic Linux commands

The following table lists a few basic Linux commands:

Command	Description
`$ ls`	This command is used to check the content of the directory.
`$ pwd`	This command is used to check the present working directory.
`$ mkdir work`	We will work in a separate directory called `work` in our `home` directory. Use this command to create a new directory called `work` in the current folder.
`$ cd work`	This command will change our working directory to the newly created `work` directory.
`$ pwd`	This command can be used to verify whether we moved to the expected directory.
`$ touch hello.sh`	This command is used to create a new empty file called `hello.sh` in the current folder.
`$ cp hello.sh bye.sh`	This command is used to copy one file into another file. This will copy `hello.sh` as `bye.sh`.

`$ mv bye.sh welcome.sh`	This command is used to rename a file. This will rename `bye.sh` as `welcome.sh`.
`$ ll`	This command will display detailed information about files.
`$ mv welcome.sh .welcome.sh` `$ ls`	Let's see some magic. Rename the file using the `mv` command and run the `ls` command. Now, the `ls` command will not display our file `.welcome.sh`. The file is hidden. Any file or directory name starting with `.` (dot) becomes hidden.
`$ ls -a`	This command is used to display hidden files.
`$ rm .welcolme.sh`	This command is used to delete the file.

If we delete any file from the GUI, such as the GUI, then it will be moved to the `/home/user/.local/share/Trash/files/` all deleted files folder.

Our first script – Hello World

Since we have learned basic commands in the Linux OS, we will now write our first shell script called `hello.sh`. You can use any editor of your choice, such as vi, gedit, nano, emacs, geany, and other similar editors. I prefer to use the vi editor:

1. Create a new `hello.sh` file as follows:

```
#!/bin/bash
# This is comment line
echo "Hello World"
ls
date
```

2. Save the newly created file.

The `#!/bin/bash` line is called the shebang line. The combination of the characters `#` and `!` is called the magic sequence. The shell uses this to call the intended shell, such as `/bin/bash` in this case. This should always be the first line in a shell script.

The next few lines in the shell script are self-explanatory:

- Any line starting with `#` will be treated as a comment line. An exception to this would be the first line with `#!/bin/bash`
- The `echo` command will print `Hello World` on the screen
- The `ls` command will display directory content in the console
- The `date` command will show the current date and time

We can execute the newly created file with the following commands:

- Technique one:

```
$ bash hello.sh
```

- Technique two:

```
$ chmod +x hello.sh
```

By running any of the preceding commands, we are adding executable permissions to our newly created file. You will learn more about file permissions later in this chapter:

```
$ ./hello.sh
```

By running the preceding command, we are executing `hello.sh` as the executable file. With technique one, we passed a filename as an argument to the bash shell.

The output of executing `hello.sh` will be as follows:

```
Hello World
hello.sh
Sun Jan 18 22:53:06 IST 2015
```

Since we have successfully executed our first script, we will proceed to develop a more advanced script, `hello1.sh`. Please create the new `hello.sh` script as follows:

```
#!/bin/bash
# This is the first Bash shell
# Scriptname : Hello1.sh
# Written by:  Ganesh Naik
echo "Hello $LOGNAME, Have a nice day !"
echo "You are working in directory `pwd`."
echo "You are working on a machine called `uname -o`."
echo "List of files in your directory is :"
ls      # List files in the present working directory
echo  "Bye for now $LOGNAME. The time is `date +%T`!"
```

The output of executing `hello.sh` will be as follows:

```
Hello student, Have a nice day !.
Your are working in directory /home/student/work.
You are working on a machine called GNU/Linux.
List of files in your directory is :
hello1.sh  hello.sh
Bye for now student. The time is 22:59:03!
```

You will learn about the `LOGNAME`, `uname`, and other similar commands as we go through the book.

Compiler and interpreter – differences in process

In any program development, the following are the two options:

- **Compilation**: Using a compiler-based language, such as C, C++, Java, and other similar languages
- **Interpreter**: Using interpreter-based languages, such as Bash shell scripting.

When we use a compiler-based language, we compile the complete source code and, as a result of compilation, we get a binary executable file. We then execute the binary to check the performance of our program.

On the other hand, when we develop the shell script, such as an interpreter-based program, every line of the program is input to the Bash shell. The lines of shell script are executed one by one sequentially. Even if the second line of a script has an error, the first line will be executed by the shell interpreter.

When not to use scripts

Shell scripts have certain advantages over compiler-based programs, such as C or C++ language. However, shell scripting has certain limitations as well.

The following are the advantages:

- Scripts are easy to write
- Scripts are quick to start and easy for debugging
- They save time in development

- Tasks of administration are automated
- No additional setup or tools are required for developing or testing shell scripts

The following are the limitations of shell scripts:

- Every line in shell script creates a new process in the operating system. When we execute the compiled program, such as a C program, it runs as a single process for the complete program.
- Since every command creates a new process, shell scripts are slow compared to compiled programs.
- Shell scripts are not suitable if heavy math operations are involved.
- There are problems with cross-platform portability.

We cannot use shell scripts in the following situations:

- Where extensive file operations are required
- Where we need data structures, such as linked lists or trees
- Where we need to generate or manipulate graphics or GUIs
- Where we need direct access to system hardware
- Where we need a port or socket I/O
- Where we need to use libraries or interface with legacy code
- Where proprietary, closed source applications are used (shell scripts put the source code right out in the open for the entire world to see)

Various directories

We will explore the directory structure in Linux so that it will be useful later on:

- `/bin/`: This contains commands used by a regular user.
- `/boot/`: The files required for the operating system startup are stored here.
- `/cdrom/`: When a CD-ROM is mounted, the CD-ROM files are accessible here.
- `/dev/`: The device driver files are stored in this folder. These device driver files will point to hardware-related programs running in the kernel.
- `/etc/`: This folder contains configuration files and startup scripts.

- `/home/`: This folder contains a home folder of all users, except the administrator.
- `/lib/`: The library files are stored in this folder.
- `/media/`: External media, such as a USB pen drive, are mounted in this folder.
- `/opt/`: The optional packages are installed in this folder.
- `/proc/`: This contains files that give information about the kernel and every process running in the OS.
- `/root/`: This is the administrator's home folder.
- `/sbin/`: This contains commands used by the administrator or root user.
- `/usr/`: This contains secondary programs, libraries, and documentation about user-related programs.
- `/var/`: This contains variable data, such as HTTP, TFTP, logs, and others.
- `/sys/`: This dynamically creates the `sys` files.

Working more effectively with Shell – basic commands

Let's learn a few commands that are required very often, such as `man`, `echo`, `cat`, and similar:

- Enter the following command. It will show the various types of manual pages displayed by the `man` command:

```
$ man man
```

- From the following table, you can get an idea about various types of `man` pages for the same command:

Section number	Subject area
1	User commands
2	System calls
3	Library calls
4	Special files
5	File formats
6	Games
7	Miscellaneous

8 System admin
9 Kernel routines

- We can enter the `man` command to display the corresponding manual pages as follows:

```
$ man 1 command
$ man 5 command
```

- Suppose we need to know more about the `passwd` command, which is used for changing the current password of a user. You can type the command as follows:

```
$ man command
  man -k passwd   // show all pages with keyword
  man -K passwd  // will search all manual pages content for pattern
"passwd"
$ man passwd
```

- This will show information about the `passwd` command:

```
$ man 5 passwd
```

- The preceding command will give information about the file `passwd`, which is stored in the `/etc/` folder, such as `/etc/passwd`.

- We can get brief information about the command as follows:

```
$ whatis passwd
```

- Output:

```
passwd (1ssl)          - compute password hashes
passwd (1)             - change user password
passwd (5)             - the password file
```

- Every command we type in the Terminal has an executable binary program file associated with it. We can check the location of a binary file as follows:

```
$ which passwd
/usr/bin/passwd
```

- The preceding line tells us that the binary file of the `passwd` command is located in the `/usr/bin/passwd` folder.

- We can get complete information about the binary file location, as well as the manual page location of any command, with the following:

```
$ whereis passwd
```

- The output will be as follows:

```
    passwd: /usr/bin/passwd /etc/passwd /usr/bin/X11/passwd
/usr/share/man/man1/passwd.1.gz /usr/share/man/man1/passwd.1ssl.gz
/usr/share/man/man5/passwd.5.gz
```

- Change the user login and effective username:

```
$ whoami
```

- This command displays the username of the logged in user:

```
$ su
```

- The `su` (switch user) command will make the user the administrator but you should know the administrator's password. The `sudo` (superuser do) command will run the command with administrator privileges. The user should have been added to the `sudoers` list.

```
# who am i
```

- This command will show the effective user who is working at that moment.

```
# exit
```

- Many a times, you might need to create new commands from existing commands. Sometimes, existing commands have complex options to remember. In such cases, we can create new commands as follows:

```
$ alias ll='ls -l'
$ alias copy='cp -rf'
```

- To list all declared aliases, use the following command:

```
$ alias
```

- To remove an alias, use the following command:

```
$ unalias copy
```

- We can check operating system details, such as UNIX/Linux or the distribution that is installed with the following command:

```
$ uname
```

- Output:

```
Linux
```

This will display the basic OS information (Unix name)

- Linux kernel version information will be displayed by the following:

```
$ uname -r
```

- Output:

```
3.13.0-32-generic
```

- To get all the information about a Linux machine, use the following command:

```
$ uname -a
```

- Output:

```
Linux localhost.localdomain 3.10.0-693.el7.x86_64 #1 SMP Tue Aug 22
21:09:27 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

- The following commands will give you more information about the Linux distribution:

```
$ cat /proc/version   // detailed info about distribution
$ cat /etc/*release
# lsb_release -a
.
```

```
[student@localhost ~]$ lsb_release -a
LSB Version:    :core-4.1-amd64:core-4.1-noarch:cxx-4.1-amd64:cxx-4.1-noarch:desktop
-4.1-amd64:desktop-4.1-noarch:languages-4.1-amd64:languages-4.1-noarch:printing-4.1-
amd64:printing-4.1-noarch
Distributor ID: CentOS
Description:    CentOS Linux release 7.4.1708 (Core)
Release:        7.4.1708
Codename:       Core
```

- The `cat` command is used for reading files and is displayed on the standard output.

- Sometimes, we need to copy a file or directory to many places. In such situations, instead of copying the original file or directory again and again, we can create soft links. In Windows, it is a similar feature to creating a shortcut.

```
$ ln -s file file_link
```

- To learn about the type of file, you can use the command file. In Linux, various types of file exist. Some examples are as follows:
    - Regular file (`-`)
    - Directory (`d`)
    - Soft link (`l`)
    - Character device driver (`c`)
    - Block device driver (`b`)
    - Pipe file (`p`)
    - Socket file (`s`)
- We can get information about a file using the following command:

```
$ file file_name
```

- Printing some text on screen for showing results to the user, or to ask for details is an essential activity.
- The following command will create a new file called `file_name` using the `cat` command:

```
$ cat > file_name
line 1
line 2
line 3
< Cntrl + D will save the file    >
```

- But this is very rarely used, as many powerful editors already exist, such as vi or gedit.
- The following command will print `Hello World` on the console. The `echo` command is very useful for shell script writers:

```
$ echo "Hello World"
```

```
$ echo "Hello World" > hello.sh
```

- The `echo` command with `>` overwrites the content of the file. If the content already exists in the file, it will be deleted and new content added. In situations where we need to append the text to the file, then we can use the `echo` command as follows:

```
$ echo "Hello World" >> hello.sh  will append the text
```

- The following command will copy the `Hello World` string to the `hello.sh` file:
- The following command will display the content of the file on screen:

```
$ cat hello.sh
```

Working with permissions

The following are the types of permissions:

- **Read permission**: The user can read or check the content of the file
- **Write permission**: The user can edit or modify the file
- **Execute permission**: The user can execute the file

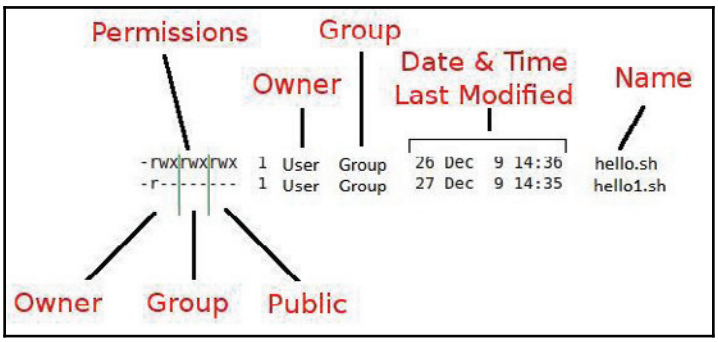
Changing file permissions

The following are the commands for changing file permissions:

To check the file permission, enter the following command:

```
$ ll file_name
```

The file permission details are as seen in the following diagram:



In the preceding diagram, as we can see, permissions are grouped in owner-user, group, and other users' permissions. Permissions are of three types–read, write, and execute. As per the requirement, we may need to change the permissions of the various files.

The chmod command

We can change the file or directory permissions in the following two ways:

Technique one – the symbolic method

The following command will add the read/write and execute permissions to the file wherein `u` is for user, `g` is for group, and `o` is for others:

```
$ chmod ugo+rwx file_name
```

Alternatively, you can use the following command:

```
$ chmod +rwx file_name
```

Technique two – the numeric method

The following command will change the file permissions using the octal technique:

```
$ chmod 777 file_name
```

The file permission `777` can be understood as `111 111 111`, which corresponds to the `rwx.rwx.rwx` permissions.

Setting umask

We will see how Linux decides the default permissions of the newly created file or folder:

```
$ umask
0002
```

The meaning of the preceding output is that, if we create a new directory, then, from the permissions of `+rwx`, the permission `0002` will be subtracted. This means that for a newly created directory, the permissions will be `775`, or `rwx rwx r-x`. For a newly created file, the file permissions will be `rw- rw- r--`. By default, for any newly created text file, the execute bit will never be set. Therefore, the newly created text file and the directory will have different permissions, even though `umask` is the same.

Setuid

Another very interesting functionality is the `setuid` feature. If the `setuid` bit is set for a script, then the script will always run with the owner's privileges, irrespective of which user is running the script. If the administrator wants to run a script written by him by other users, then he can set this bit.

Consider either of the following situations:

```
$ chmod u+s file_name
$ chmod 4777 file
```

The file permissions after any of the preceding two commands will be `drwsrwxrwx`.

Setgid

Similar to `setuid`, the `setgid` functionality gives the user the ability to run scripts with a group owner's privileges, even if it is executed by any other user:

```
$ chmod g+s filename
```

Alternatively, you can use the following command:

```
$ chmod 2777 filename
```

File permissions after any of the preceding two commands will be `drwxrwsrwtx`.

Sticky bit

The sticky bit is a very interesting functionality. Let's say, in the administration department, there are 10 users. If one folder has been set with sticky bit, then all other users can copy files to that folder. All users can read the files, but only the owner of the respective file can edit or delete the file. Other users can only read, but not edit or modify, the files if the sticky bit is set:

```
$ chmod +t filename
```

Alternatively, you can use the following command:

```
$ chmod 1777
```

File permissions after any of the preceding two commands will be `drwxrwxrwt`.

Summary

In this chapter, you learned different ways to write and run shell scripts. You also learned ways to handle files and directories, as well as work with permissions.

In the next chapter, you will learn about process management, job control, and automation.